# Part V

## Chapter List



# Chapter 13: Graphs

## Overview

Graphs are one of the most versatile structures used in computer programming. The sorts of problems that graphs can help solve are generally quite different from those we've dealt with thus far in this book. If you're dealing with general kinds of data storage problems, you probably won't need a graph, but for some problems—and they tend to be interesting ones—a graph is indispensable.

Our discussion of graphs is divided into two chapters. In this chapter we'll cover the algorithms associated with unweighted graphs, show some algorithms that these graphs can represent, and present two Workshop applets to model them. In the next chapter we'll look at the more complicated algorithms associated with weighted graphs.

## Introduction to Graphs

Graphs are data structures rather like trees. In fact, in a mathematical sense, a tree is a kind of graph. In computer programming, however, graphs are used in different ways than trees.

The data structures examined previously in this book have an architecture dictated by the algorithms used on them. For example, a binary tree is shaped the way it is because that shape makes it easy to search for data and insert new data. The edges in a tree represent quick ways to get from node to node.

Graphs, on the other hand, often have a shape dictated by a physical problem. For example, nodes in a graph may represent cities, while edges may represent airline flight routes between the cities. Another more abstract example is a graph representing the individual tasks necessary to complete a project. In the graph, nodes may represent tasks, while directed edges (with an arrow at one end) indicate which task must be completed before another. In both cases, the shape of the graph arises from the specific real-world situation.

Before going further, we must mention that, when discussing graphs, nodes are called *vertices* (the singular is *vertex*). This is probably because the nomenclature for graphs is older than that for trees, having arisen in mathematics centuries ago. Trees are more closely associated with computer science.

## Definitions

Figure 13.1-a shows a simplified map of the freeways in the vicinity of San Jose, California. Figure 13.1-b shows a graph that models these freeways.

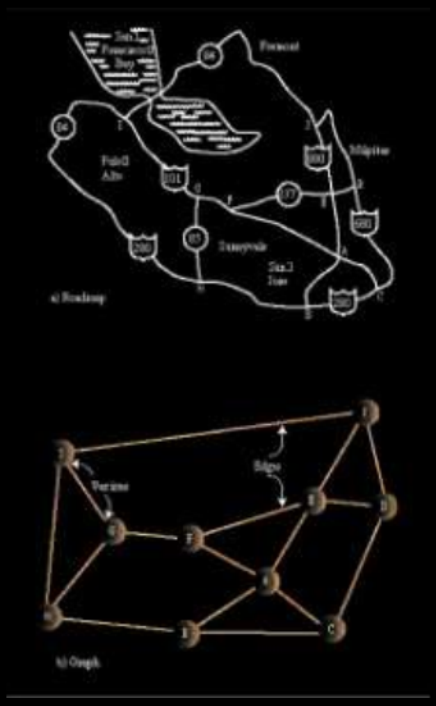

**Figure 13.1:** Road map and graph

In the graph, circles represent freeway interchanges and straight lines connecting the circles represent freeway segments. The circles are *vertices*, and the lines are *edges*. The vertices are usually labeled in some way—often, as shown here, with letters of the alphabet. Each edge is bounded by the two vertices at its ends.

The graph doesn't attempt to reflect the geographical positions shown on the map; it shows only the relationships of the vertices and the edges—that is, which edges are connected to which vertex. It doesn't concern itself with physical distances or directions. Also, one edge may represent several different route numbers, as in the case of the edge from I to H, which involves routes 101, 84, and 280. It's the *connectedness* (or lack of it) of one intersection to another that's important, not the actual routes.

## Adjacency

Two vertices are said to be *adjacent* to one another if they are connected by a single edge. Thus in Figure 13.1, vertices I and G are adjacent, but vertices I and F are not. The vertices adjacent to a given vertex are sometimes said to be its *neighbors*. For example, the neighbors of G are I, H, and F.

## Paths

A *path* is a sequence of edges. Figure 13.1 shows a path from vertex B to vertex J that passes through vertices A and E. We can call this path BAEJ. There can be more than one path between two vertices; another path from B to J is BCDJ.

## Connected Graphs

A graph is said to be *connected* if there is at least one path from every vertex to every other vertex, as in the graph in Figure 13.2-a. However, if "You can't get there from here" (as Vermont farmers traditionally tell city slickers who stop to ask for directions), the

graph is not connected, as in Figure 13.2-b.

A non-connected graph consists of several *connected components*. In Figure 13.2-b, A and B are one connected component, and C and D are another.

For simplicity, the algorithms we'll be discussing in this chapter are written to apply to connected graphs, or to one connected component of a non-connected graph. If appropriate, small modifications will usually enable them to work with non-connected graphs as well.

### Directed and Weighted Graphs

The graphs in Figures 13.1 and 13.2 are *non-directed* graphs. That means that the edges don't have a *direction*; you can go either way on them. Thus you can go from vertex A to vertex B, or from vertex B to vertex A, with equal ease. (This models freeways appropriately, because you can usually go either way on a freeway.)

However, graphs are often used to model situations in which you can go in only one direction along an edge; from A to B but not from B to A, as on a one-way street. Such a graph is said to be *directed*. The allowed direction is typically shown with an arrowhead at the end of the edge.

In some graphs, edges are given a *weight*, a number that can represent the physical distance between two vertices, or the time it takes to get from one vertex to another, or how much it costs to travel from vertex to vertex (on airline routes, for example). Such graphs are called *weighted* graphs. We'll explore them in the next chapter.

We're going to begin this chapter by discussing simple undirected, unweighted graphs; later we'll explore directed unweighted graphs.

We have by no means covered all the definitions that apply to graphs; we'll introduce more as we go along.

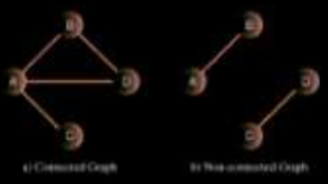

**Figure 13.2:** onnected and nonconnected graphs

## Historical Note

One of the first mathematicians to work with graphs was Leonhard Euler in the early 18th century. He solved a famous problem dealing with the bridges in the town of Königsberg, Poland. This town included an island and seven bridges, as shown in Figure 13.3-a.

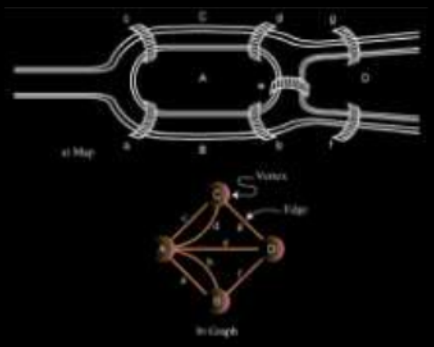

**Figure 13.3:** The bridges of Königsberg

The problem, much discussed by the townsfolk, was to find a way to walk across all seven bridges without recrossing any of them. We won't recount Euler's solution to the problem; it turns out that there is no such path. However, the key to his solution was to represent the problem as a graph, with land areas as vertices and bridges as edges, as shown in Figure 13.3-b. This is perhaps the first example of a graph being used to represent a problem in the real world.

## Representing a Graph in a Program

It's all very well to think about graphs in the abstract, as Euler and other mathematicians did until the invention of the computer, but we want to represent graphs by using a computer. What sort of software structures are appropriate to model a graph? We'll look at vertices first, and then at edges.

### Vertices

In a very abstract graph program you could simply number the vertices 0 to N–1 (where N is the number of vertices). You wouldn't need any sort of variable to hold the vertices, because their usefulness would result from their relationships with other vertices.

In most situations, however, a vertex represents some real-world object, and the object must be described using data items. If a vertex represents a city in an airline route simulation, for example, it may need to store the name of the city, its altitude, its location, and other such information. Thus it's usually convenient to represent a vertex by an object of a vertex class. Our example programs store only a letter (like A), used as a label for identifying the vertex, and a flag for use in search algorithms, as we'll see later. Here's how the `Vertex` class looks:

```
class Vertex
   {
   public char label;        // label (e.g. 'A')
   public boolean wasVisited;

   public Vertex(char lab)   // constructor
      {
      label = lab;
      wasVisited = false;
      }

   }  // end class Vertex
```

Vertex objects can be placed in an array and referred to using their index number. In our

examples we'll store them in an array called `vertexList`. The vertices might also be placed in a list or some other data structure. Whatever structure is used, this storage is for convenience only. It has no relevance to how the vertices are connected by edges. For this, we need another mechanism.

## Edges

In Chapter 9, "Red-Black Trees," we saw that a computer program can represent trees in several ways. Mostly we examined trees in which each node contained references to its children, but we also mentioned that an array could be used, with a node's position in the array indicating its relationship to other nodes. Chapter 12, "Heaps," described arrays used to represent a kind of tree called a *heap*.

A graph, however, doesn't usually have the same kind of fixed organization as a tree. In a binary tree, each node has a maximum of two children, but in a graph each vertex may be connected to an arbitrary number of other vertices. For example, in Figure 13.2-a, vertex A is connected to three other vertices, whereas C is connected to only one.

To model this sort of free-form organization, a different approach to representing edges is preferable to that used for trees. Two methods are commonly used for graphs:the *adjacency matrix* and the *adjacency list*. (Remember that one vertex is said to be *adjacent* to another if they're connected by a single edge.)

## The Adjacency Matrix

An adjacency matrix is a two-dimensional array in which the elements indicate whether an edge is present between two vertices. If a graph has N vertices, the adjacency matrix is an NxN array. Table 13.1 shows the adjacency matrix for the graph in Figure 13.2-a.

**Table 13.1: Adjacency Matrix**

| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 1 | 1 | 1 |
| B | 1 | 0 | 0 | 1 |
| C | 1 | 0 | 0 | 0 |
| D | 1 | 1 | 0 | 0 |

The vertices are used as headings for both rows and columns. An edge between two vertices is indicated by a 1; the absence of an edge is a 0. (You could also use Boolean `true`/`false` values.) As you can see, vertex A is adjacent to all three other vertices, B is adjacent to A and D, C is adjacent only to A, and D is adjacent to A and B. In this example, the "connection" of a vertex to itself is indicated by 0, so the diagonal from upper-left to lower-right, A-A to D-D, which is called the *identity diagonal*, is all 0s. The entries on the identity diagonal don't convey any real information, so you can equally well put 1s along it, if that's more convenient in your program.

Note that the triangular-shaped part of the matrix above the identity diagonal is a mirror

image of the part below; both triangles contain the same information. This redundancy may seem inefficient, but there's no convenient way to create a triangular array in most computer languages, so it's simpler to accept the redundancy. Consequently, when you add an edge to the graph, you must make two entries in the adjacency matrix rather than one.

### The Adjacency List

The other way to represent edges is with an adjacency list. The *list* in *adjacency list* refers to a linked list of the kind we examined in Chapter 6, "Recursion." Actually, an adjacency list is an array of lists (or a list of lists). Each individual list shows what vertices a given vertex is adjacent to. Table 13.2 shows the adjacency lists for the graph of Figure 13.2-a.

**Table 13.2: Adjacency Lists**

| Vertex | List Containing Adjacent Vertices |
|---|---|
| A | B → C → D |
| B | A → D |
| C | A |
| D | A → B |

In this table, the → symbol indicates a link in a linked list. Each link in the list is a vertex. Here the vertices are arranged in alphabetical order in each list, although that's not really necessary. Don't confuse the contents of adjacency lists with paths. The adjacency list shows which vertices are adjacent to—that is, one edge away from—a given vertex, not paths from vertex to vertex.

Later we'll discuss when to use an adjacency matrix as opposed to an adjacency list. The workshop applets shown in this chapter all use the adjacency matrix approach, but sometimes the list approach is more efficient.

## Adding Vertices and Edges to a Graph

To add a vertex to a graph, you make a new vertex object with `new` and insert it into your vertex array, `vertexList`. In a real-world program a vertex might contain many data items, but for simplicity we'll assume that it contains only a single character. Thus the creation of a vertex looks something like this:

```
vertexList[nVerts++] = new Vertex('F');
```

This inserts a vertex F, where `nVerts` is the number of vertices currently in the graph.

How you add an edge to a graph depends on whether you're using an adjacency matrix or adjacency lists to represent the graph. Let's say that you're using an adjacency matrix and want to add an edge between vertices 1 and 3. These numbers correspond to the

array indices in `vertexList` where the vertices are stored. When you first created the adjacency matrix `adjMat`, you filled it with 0s. To insert the edge, you say

```
adjMat[1][3] = 1;
adjMat[3][1] = 1;
```

If you were using an adjacency list, you would add a 1 to the list for 3, and a 3 to the list for 1.

## The Graph Class

Let's look at a class `Graph` that contains methods for creating a vertex list and an adjacency matrix, and for adding vertices and edges to a `Graph` object:

```
class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // array of vertices
   private int adjMat[][];      // adjacency matrix
   private int nVerts;          // current number of vertices

// -------------------------------------------------------------
-
   public Graph()               // constructor
      {
      vertexList = new Vertex[MAX_VERTS];
                                          // adjacency matrix
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // set adjacency
         for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
            adjMat[j][k] = 0;
      }  // end constructor

// -------------------------------------------------------------
-
   public void addVertex(char lab)    // argument is label
      {
      vertexList[nVerts++] = new Vertex(lab);
      }

// -------------------------------------------------------------
-
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      adjMat[end][start] = 1;
      }

// -------------------------------------------------------------
-
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
```

```
        }

    // ---------------------------------------------------------
    -

        }  // end class Graph
```

Within the `Graph` class, vertices are identified by their index number in `vertexList`.

We've already discussed most of the methods shown here. To display a vertex, we simply print out its one-character label.

The adjacency matrix (or the adjacency list) provides information that is local to a given vertex. Specifically, it tells you which vertices are connected by a single edge to a given vertex. To answer more global questions about the arrangement of the vertices, we must resort to various algorithms. We'll begin with searches.

## Searches

One of the most fundamental operations to perform on a graph is finding which vertices can be reached from a specified vertex. For example, imagine trying to find out how many towns in the United States can be reached by passenger train from Kansas City (assuming that you don't mind changing trains). Some towns could be reached. Others couldn't be reached because they didn't have passenger rail service. Possibly others couldn't be reached, even though they had rail service, because their rail system (the narrow-gauge Hayfork-Hicksville RR, for example) didn't connect with the standard-gauge line you started on or any of the lines that could be reached from your line.

Here's another situation in which you might need to find all the vertices reachable from a specified vertex. Imagine that you're designing a printed circuit board, like the ones inside your computer. (Open it up and take a look!) Various components—mostly integrated circuits (ICs)—are placed on the board, with pins from the ICs protruding through holes in the board. The ICs are soldered in place, and their pins are electrically connected to other pins by *traces*—thin metal lines applied to the surface of the circuit board, as shown in Figure 13.4. (No, you don't need to worry about the details of this figure.)

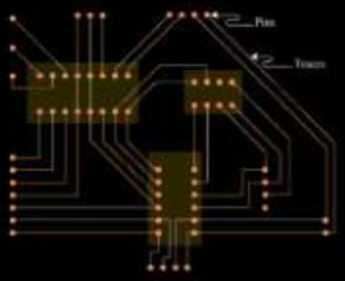

**Figure 13.4:** ins and traces on a circuit boardSearches

In a graph, each pin might be represented by a vertex, and each trace by an edge. On a circuit board there are many electrical circuits that aren't connected to each other, so the graph is by no means a connected one. During the design process, therefore, it may be genuinely useful to create a graph and use it to find which pins are connected to the same electrical circuit.

Assume that you've created such a graph. Now you need an algorithm that provides a systematic way to start at a specified vertex, and then move along edges to other vertices, in such a way that when it's done you are guaranteed that it has *visited* every

vertex that's connected to the starting vertex. Here, as it did in Chapter 8, "Binary Trees," when we discussed binary trees, *visit* means to perform some operation on the vertex, such as displaying it.

There are two common approaches to searching a graph: *depth-first search (DFS)* and *breadth-first search (BFS)*. Both will eventually reach all connected vertices. The difference is that the depth-first search is implemented with a stack, whereas the breadth-first search is implemented with a queue. These mechanisms result, as we'll see, in the graph being searched in different ways.

## Depth-First Search

The depth-first search uses a stack to remember where it should go when it reaches a dead end. We'll show an example, encourage you to try similar examples with the GraphN Workshop applet, and then finally show some code that carries out the search.

### An Example

We'll discuss the idea behind the depth-first search in relation to Figure 13.5. The numbers in this figure show the order in which the vertices are visited.

**Figure 13.5:** Depth-first search

To carry out the depth-first search, you pick a starting point—in this case, vertex A. You then do three things: visit this vertex, push it onto a stack so you can remember it, and mark it so you won't visit it again.

Next you go to any vertex adjacent to A that hasn't yet been visited. We'll assume the vertices are selected in alphabetical order, so that brings up B. You visit B, mark it, and push it on the stack.

Now what? You're at B, and you do the same thing as before: go to an adjacent vertex that hasn't been visited. This leads you to F. We can call this process Rule 1.

**REMEMBER**
Rule 1: If possible, visit an adjacent unvisited vertex, mark it, and push it on the stack.

Applying Rule 1 again leads you to H. At this point, however, you need to do something else, because there are no unvisited vertices adjacent to H. Here's where Rule 2 comes in.

**REMEMBER**
Rule 2: If you can't follow Rule 1, then, if possible, pop a vertex off the stack.

Following this rule, you pop H off the stack, which brings you back to F. F has no unvisited adjacent vertices, so you pop it. Ditto B. Now only A is left on the stack.

A, however, does have unvisited adjacent vertices, so you visit the next one, C. But C is the end of the line again, so you pop it and you're back to A. You visit D, G, and I, and then pop them all when you reach the dead end at I. Now you're back to A. You visit E,

and again you're back to A.

This time, however, A has no unvisited neighbors, so we pop it off the stack. But now there's nothing left to pop, which brings up Rule 3.

**REMEMBER**
Rule 3: If you can't follow Rule 1 or Rule 2, you're finished.

Table 13.3 shows how the stack looks in the various stages of this process, as applied to Figure 13.5.

**Table 13.3: Stack Contents During Depth-First Search**

| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |
| Pop F | AB |
| Pop B | A |
| Visit C | AC |
| Pop C | A |
| Visit D | AD |
| Visit G | ADG |
| Visit I | ADGI |
| Pop I | ADG |
| Pop G | AD |
| Pop D | A |
| Visit E | AE |
| Pop E | A |

```
Pop A

Done
```

The contents of the stack is the route you took from the starting vertex to get where you are. As you move away from the starting vertex, you push vertices as you go. As you move back toward the starting vertex, you pop them. The order in which you visit the vertices is ABFHCDGIE.

You might say that the depth-first search algorithm likes to get as far away from the starting point as quickly as possible, and returns only when it reaches a dead end. If you use the term *depth* to mean the distance from starting point, you can see where the name *depth-first search* comes from.

## An Analogy

An analogy you might think about in relation to depth-first search is a maze. The maze—perhaps one of the people-size ones made of hedges, popular in England—consists of narrow passages (think of edges) and intersections where passages meet (vertices).

Suppose that someone is lost in the maze. She knows there's an exit and plans to traverse the maze systematically to find it. Fortunately, she has a ball of string and a marker pen. She starts at some intersection and goes down a randomly chosen passage, unreeling the string. At the next intersection, she goes down another randomly chosen passage, and so on, until finally she reaches a dead end.

At the dead end she retraces her path, reeling in the string, until she reaches the previous intersection. Here she marks the path she's been down so she won't take it again, and tries another path. When she's marked all the paths leading from that intersection, she returns to the previous intersection and repeats the process.

The string represents the stack: It "remembers" the path taken to reach a certain point.

## The GraphN Workshop Applet and DFS

You can try out the depth-first search with the DFS button in the GraphN Workshop applet. (The N is for *not directed, not weighted*.)

Start the applet. At the beginning, there are no vertices or edges, just an empty rectangle. You create vertices by double-clicking the desired location. The first vertex is automatically labeled A, the second one is B, and so on. They're colored randomly.

To make an edge, drag from one vertex to another. Figure 13.6 shows the graph of Figure 13.5 as it looks when created using the applet.

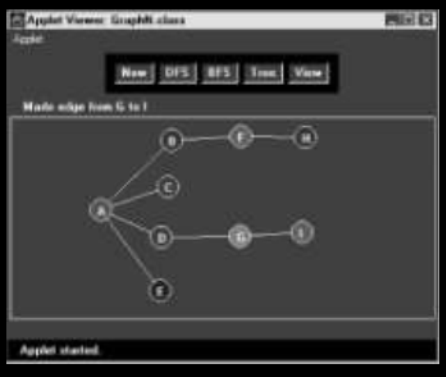

**Figure 13.6:** The GraphN Workshop applet

There's no way to delete individual edges or vertices, so if you make a mistake, you'll need to start over by clicking the New button, which erases all existing vertices and edges. (It warns you before it does this.) Clicking the View button switches you to the adjacency matrix for the graph you've made, as shown in Figure 13.7. Clicking View again switches you back to the graph.

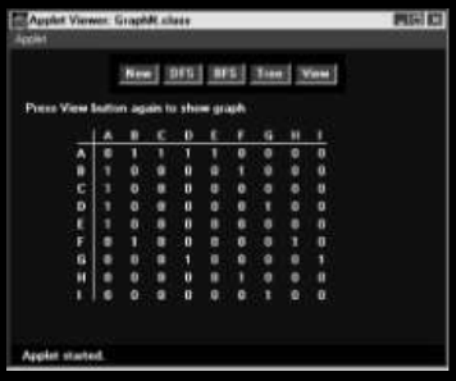


**Figure 13.7:** Adjacency matrix view in GraphNSearches

To run the depth-first search algorithm, click the DFS button repeatedly. You'll be prompted to click (*not* double-click) the starting vertex at the beginning of the process.

You can re-create the graph of Figure 13.6, or you can create simpler or more complex ones of your own. After you play with it a while, you can predict what the algorithm will do next (unless the graph is too weird).

If you use the algorithm on an unconnected graph, it will find only those vertices that are connected to the starting vertex.

## Java Code

A key to the DFS algorithm is being able to find the vertices that are unvisited and adjacent to a specified vertex. How do you do this? The adjacency matrix is the key. By going to the row for the specified vertex and stepping across the columns, you can pick out the columns with a 1; the column number is the number of an adjacent vertex. You can then check whether this vertex is unvisited. If so, you've found what you want—the next vertex to visit. If no vertices on the row are simultaneously 1 (adjacent) and also unvisited, then there are no unvisited vertices adjacent to the specified vertex. We put the code for this process in the `getAdjUnvisitedVertex()` method:

```
// returns an unvisited vertex adjacent to v
public int getAdjUnvisitedVertex(int v)
   {
   for(int j=0; j<nVerts; j++)
      if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
         return j;               // return first such vertex
   return -1;                    // no such vertices

   }  // end getAdjUnvisitedVert()
```

Now we're ready for the `dfs()` method of the `Graph` class, which actually carries out the depth-first search. You can see how this code embodies the three rules listed earlier. It loops until the stack is empty. Within the loop, it does four things:

1. It examines the vertex at the top of the stack, using `peek()`.

2. It tries to find an unvisited neighbor of this vertex.

3. If it doesn't find one, it pops the stack.

4. If it finds such a vertex, it visits it and pushes it onto the stack.

Here's the code for the `dfs()` method:

```
public void dfs()  // depth-first search
   {                                 // begin at vertex 0
   vertexList[0].wasVisited = true;  // mark it
   displayVertex(0);                 // display it
   theStack.push(0);                 // push it

   while( !theStack.isEmpty() )      // until stack empty,
      {
      // get an unvisited vertex adjacent to stack top
      int v = getAdjUnvisitedVertex( theStack.peek() );
      if(v == -1)                    // if no such vertex,
         theStack.pop();             //    pop a new one
      else                           // if it exists,
         {
         vertexList[v].wasVisited = true;  // mark it
         displayVertex(v);                 // display it
         theStack.push(v);                 // push it
         }
      }  // end while

      // stack is empty, so we're done
      for(int j=0; j<nVerts; j++)    // reset flags
         vertexList[j].wasVisited = false;

   }  // end dfs
```

At the end of `dfs()`, we reset all the `wasVisited` flags so we'll be ready to run `dfs()` again later. The stack should already be empty, so it doesn't need to be reset.

Now we have all the pieces of the `Graph` class we need. Here's some code that creates a graph object, adds some vertices and edges to it, and then performs a depth-first search:

```
Graph theGraph = new Graph();
theGraph.addVertex('A');    // 0  (start for dfs)
theGraph.addVertex('B');    // 1
theGraph.addVertex('C');    // 2
theGraph.addVertex('D');    // 3
theGraph.addVertex('E');    // 4
```

```
theGraph.addEdge(0, 1);     // AB
theGraph.addEdge(1, 2);     // BC
theGraph.addEdge(0, 3);     // AD
theGraph.addEdge(3, 4);     // DE

System.out.print("Visits: ");
theGraph.dfs();             // depth-first search
System.out.println();
```

Figure 13.8 shows the graph created by this code. Here's the output:

```
Visits: ABCDE
```

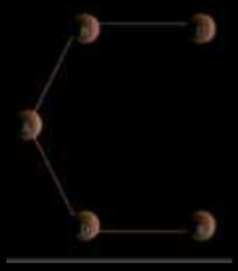

**Figure 13.8:** Graph used by `dfs.java` and `bfs.javaSearches`

You can modify this code to create the graph of your choice, and then run it to see it carry out the depth-first search.

## The `dfs.java` Program

Listing 13.1 shows the `dfs.java` program, which includes the `dfs()` method. It includes a version of the `StackX` class from Chapter 4, "Stacks and Queues."

### Listing 13.1 The dfs.java Program

```
// dfs.java
// demonstrates depth-first search
// to run this program: C>java DFSApp
import java.awt.*;
////////////////////////////////////////////////////////////////
class StackX
   {
   private final int SIZE = 20;
   private int[] st;
   private int top;
   public StackX()            // constructor
      {
      st = new int[SIZE];     // make array
      top = -1;
      }
   public void push(int j)    // put item on stack
      { st[++top] = j; }
```

```
   public int pop()          // take item off stack
      { return st[top--]; }
   public int peek()         // peek at top of stack
      { return st[top]; }
   public boolean isEmpty()  // true if nothing on stack
      { return (top == -1); }
   }  // end class StackX

////////////////////////////////////////////////////////////////

class Vertex
   {
   public char label;        // label (e.g. 'A')
   public boolean wasVisited;

// ------------------
   public Vertex(char lab)   // constructor
      {
      label = lab;
      wasVisited = false;
      }

// ------------------
   }  // end class Vertex

////////////////////////////////////////////////////////////////

class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // list of vertices
   private int adjMat[][];      // adjacency matrix
   private int nVerts;          // current number of vertices
   private StackX theStack;

// ------------------
   public Graph()               // constructor
      {
      vertexList = new Vertex[MAX_VERTS];
                                          // adjacency matrix
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // set adjacency
         for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
            adjMat[j][k] = 0;
      theStack = new StackX();
      }  // end constructor

// ------------------
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
```

```
      }

// ------------------
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      adjMat[end][start] = 1;
      }

// ------------------
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }

// ------------------
   public void dfs()  // depth-first search
      {                                 // begin at vertex 0
      vertexList[0].wasVisited = true;  // mark it
      displayVertex(0);                 // display it
      theStack.push(0);                 // push it

      while( !theStack.isEmpty() )      // until stack empty,
         {
         // get an unvisited vertex adjacent to stack top
         int v = getAdjUnvisitedVertex( theStack.peek() );
         if(v == -1)                    // if no such vertex,
            theStack.pop();
         else                           // if it exists,
            {
            vertexList[v].wasVisited = true;  // mark it
            displayVertex(v);                 // display it
            theStack.push(v);                 // push it
            }
         }  // end while

         // stack is empty, so we're done
         for(int j=0; j<nVerts; j++)          // reset flags
            vertexList[j].wasVisited = false;
      }  // end dfs

// ------------------
   // returns an unvisited vertex adj to v
   public int getAdjUnvisitedVertex(int v)
      {
      for(int j=0; j<nVerts; j++)
         if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
            return j;
      return -1;
      }  // end getAdjUnvisitedVert()

// ------------------
```

```
    }  // end class Graph

////////////////////////////////////////////////////////////////

class DFSApp
   {
   public static void main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A');    // 0  (start for dfs)
      theGraph.addVertex('B');    // 1
      theGraph.addVertex('C');    // 2
      theGraph.addVertex('D');    // 3
      theGraph.addVertex('E');    // 4

      theGraph.addEdge(0, 1);     // AB
      theGraph.addEdge(1, 2);     // BC
      theGraph.addEdge(0, 3);     // AD
      theGraph.addEdge(3, 4);     // DE

      System.out.print("Visits: ");
      theGraph.dfs();             // depth-first search
      System.out.println();
      }  // end main()
   }  // end class DFSApp

////////////////////////////////////////////////////////////////
```

## Breadth-First Search

As we saw in the depth-first search, the algorithm acts as though it wants to get as far away from the starting point as quickly as possible. In the breadth-first search, on the other hand, the algorithm likes to stay as close as possible to the starting point. It visits all the vertices adjacent to the starting vertex, and only then goes further afield. This kind of search is implemented using a queue instead of a stack.

### An Example

Figure 13.9 shows the same graph as Figure 13.5, but here the breadth-first search is used. Again, the numbers indicate the order in which the vertices are visited.

**Figure 13.9:** Breadth-first search

A is the starting vertex, so you visit it and make it the current vertex. Then you follow these rules:

**REMEMBER**

Rule 1: Visit the next unvisited vertex (if there is one) that's adjacent to the current vertex, mark it, and insert it into the queue.

**REMEMBER**

Rule 2: If you can't carry out Rule 1 because there are no more unvisited vertices, remove a vertex from the queue (if possible) and make it the current vertex.

**REMEMBER**

Rule 3: If you can't carry out Rule 2 because the queue is empty, you're finished.

Thus you first visit all the vertices adjacent to A, inserting each one into the queue as you visit it. Now you've visited A, B, C, D, and E. At this point the queue (from front to rear) contains BCDE.

There are no more unvisited vertices adjacent to A, so you remove B from the queue and look for vertices adjacent to B, so you remove C from the queue. It has no adjacent unvisited adjacent vertices, so you remove D and visit G. D has no more adjacent unvisited vertices, so you remove E. Now the queue is FG. You remove F and visit H, and then you remove G and visit I.

Now the queue is HI, but when you've removed each of these and found no adjacent unvisited vertices, the queue is empty, so you're finished. Table 13.4 shows this sequence.

**Table 13.4: Queue Contents During Breadth-First Search**

| Event | Queue (Front to Rear) |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Remove B | CDE |
| Visit F | CDEF |
| Remove C | DEF |
| Remove D | EF |
| Visit G | EFG |

| | |
|---|---|
| Remove E | FG |
| Remove F | G |
| Visit H | GH |
| Remove G | H |
| Visit I | HI |
| Remove H | I |
| Remove I | |
| Done | |

At each moment, the queue contains the vertices that have been visited but whose neighbors have not yet been fully explored. (Contrast this with the depth-first search, where the contents of the stack is the route you took from the starting point to the current vertex.) The nodes are visited in the order ABCDEFGHI.

## The GraphN Workshop Applet and BFS

Use the GraphN Workshop applet to try out a breadth-first search using the BFS button. Again, you can experiment with the graph of Figure 13.9, or you can make up your own.

Notice the similarities and the differences of the breadth-first search compared with the depth-first search.

You can think of the breadth-first search as proceeding like ripples widening when you drop a stone in water—or, for those of you who enjoy epidemiology, as the influenza virus carried by air travelers from city to city. First, all the vertices one edge (plane flight) away from the starting point are visited, then all the vertices two edges away are visited, and so on.

## Java Code

The `bfs()` method of the `Graph` class is similar to the `dfs()` method, except that it uses a queue instead of a stack and features nested loops instead of a single loop. The outer loop waits for the queue to be empty, whereas the inner one looks in turn at each unvisited neighbor of the current vertex. Here's the code:

```
public void bfs()                   // breadth-first search
   {                                // begin at vertex 0
   vertexList[0].wasVisited = true; // mark it
   displayVertex(0);                // display it
   theQueue.insert(0);              // insert at tail
   int v2;

   while( !theQueue.isEmpty() )     // until queue empty,
      {
      int v1 = theQueue.remove();   // remove vertex at head
      // until it has no unvisited neighbors
```

```
      while( (v2=getAdjUnvisitedVertex(v1)) != -1 )
         {                                  // get one,
         vertexList[v2].wasVisited = true;  // mark it
         displayVertex(v2);                 // display it
         theQueue.insert(v2);               // insert it
         }   // end while(unvisited neighbors)
      }  // end while(queue not empty)

   // queue is empty, so we're done
   for(int j=0; j<nVerts; j++)             // reset flags
      vertexList[j].wasVisited = false;

   }  // end bfs()
```

Given the same graph as in `dfs.java` (shown earlier in Figure 13.8), the output from `bfs.java` is now

```
Visits: ABDCE
```

## The `bfs.java` Program

The `bfs.java` program, shown in Listing 13.2, is similar to `dfs.java` except for the inclusion of a `Queue` class (modified from the version in Chapter 5, "Linked Lists") instead of a `StackX` class, and a `bfs()` method instead of a `dfs()` method.

### Listing 13.2 The bfs.java Program

```
// bfs.java
// demonstrates breadth-first search
// to run this program: C>java BFSApp
import java.awt.*;
////////////////////////////////////////////////////////////////
class Queue
   {
   private final int SIZE = 20;
   private int[] queArray;
   private int front;
   private int rear;

   public Queue()            // constructor
      {
      queArray = new int[SIZE];
      front = 0;
      rear = -1;
      }
   public void insert(int j) // put item at rear of queue
      {
      if(rear == SIZE-1)
         rear = -1;
      queArray[++rear] = j;
      }
   public int remove()       // take item from front of queue
      {
```

```
      int temp = queArray[front++];
      if(front == SIZE)
         front = 0;
      return temp;
      }
   public boolean isEmpty()  // true if queue is empty
      {
      return ( rear+1==front || (front+SIZE-1==rear) );
      }
   }  // end class Queue

////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // label (e.g. 'A')
   public boolean wasVisited;

// ------------------------------------------------------------
-
   public Vertex(char lab)   // constructor
      {
      label = lab;
      wasVisited = false;
      }

// ------------------------------------------------------------
-
   }  // end class Vertex

////////////////////////////////////////////////////////////////

class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // list of vertices
   private int adjMat[][];      // adjacency matrix
   private int nVerts;          // current number of vertices
   private Queue theQueue;

// -----------------
   public Graph()               // constructor
      {
      vertexList = new Vertex[MAX_VERTS];
                                          // adjacency matrix
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // set adjacency
         for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
            adjMat[j][k] = 0;
      theQueue = new Queue();
      }  // end constructor

// ------------------------------------------------------------
```

```
-
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }

// -------------------------------------------------------------
-
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      adjMat[end][start] = 1;
      }

// -------------------------------------------------------------
-
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }

// -------------------------------------------------------------
-
   public void bfs()                   // breadth-first search
      {                                // begin at vertex 0
      vertexList[0].wasVisited = true; // mark it
      displayVertex(0);                // display it
      theQueue.insert(0);              // insert at tail
      int v2;

      while( !theQueue.isEmpty() )     // until queue empty,
         {
         int v1 = theQueue.remove();   // remove vertex at head
         // until it has no unvisited neighbors
         while( (v2=getAdjUnvisitedVertex(v1)) != -1 )
            {                                 // get one,
            vertexList[v2].wasVisited = true;  // mark it
            displayVertex(v2);                 // display it
            theQueue.insert(v2);               // insert it
            }   // end while
         }  // end while(queue not empty)

      // queue is empty, so we're done
      for(int j=0; j<nVerts; j++)             // reset flags
         vertexList[j].wasVisited = false;
      }  // end bfs()

// -------------------------------------------------------------
-
   // returns an unvisited vertex adj to v
   public int getAdjUnvisitedVertex(int v)
      {
      for(int j=0; j<nVerts; j++)
         if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
```

```
            return j;
        return -1;
        }  // end getAdjUnvisitedVert()

// ------------------------------------------------------------
-
    }  // end class Graph

////////////////////////////////////////////////////////////////
class BFSApp
    {
    public static void main(String[] args)
        {
        Graph theGraph = new Graph();
        theGraph.addVertex('A');    // 0  (start for dfs)
        theGraph.addVertex('B');    // 1
        theGraph.addVertex('C');    // 2
        theGraph.addVertex('D');    // 3
        theGraph.addVertex('E');    // 4

        theGraph.addEdge(0, 1);     // AB
        theGraph.addEdge(1, 2);     // BC
        theGraph.addEdge(0, 3);     // AD
        theGraph.addEdge(3, 4);     // DE

        System.out.print("Visits: ");
        theGraph.bfs();             // breadth-first search
        System.out.println();
        }  // end main()
    }  // end class BFSApp

////////////////////////////////////////////////////////////////
```

## Minimum Spanning Trees

Suppose that you've designed a printed circuit board like the one shown in Figure 13.4, and you want to be sure you've used the minimum number of traces. That is, you don't want any extra connections between pins; such extra connections would take up extra room and make other circuits more difficult to lay out.

It would be nice to have an algorithm that, for any connected set of pins and traces (vertices and edges, in graph terminology), would remove any extra traces. The result would be a graph with the minimum number of edges necessary to connect the vertices. For example, Figure 13.10-a shows five vertices with an excessive number of edges, while Figure 13.10-b shows the same vertices with the minimum number of edges necessary to connect them. This constitutes a *minimum spanning tree*.

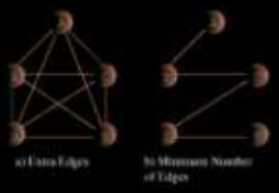

**Figure 13.10:** Minimum spanning tree

There are many possible minimum spanning trees for a given set of vertices. Figure 13.10-b shows edges AB, BC, CD, and DE, but edges AC, CE, ED, and DB would do just as well. The arithmetically inclined will note that the number of edges E in a minimum spanning tree is always one less than the number of vertices V:

$$E = V - 1$$

Remember that we're not worried here about the length of the edges. We're not trying to find a minimum physical length, just the minimum number of edges. (This will change when we talk about weighted graphs in the next chapter.)

The algorithm for creating the minimum spanning tree is almost identical to that used for searching. It can be based on either the depth-first search or the breadth-first search. In our example we'll use the depth-first-search.

It turns out that by executing the depth-first search and recording the edges you've traveled to make the search, you automatically create a minimum spanning tree. The only difference between the minimum spanning tree method `mst()`, which we'll see in a moment, and the depth-first search method `dfs()`, which we saw earlier, is that `mst()` must somehow record the edges traveled.

## GraphN Workshop Applet

Repeatedly clicking the Tree button in the GraphN Workshop algorithm will create a minimum spanning tree for any graph you create. Try it out with various graphs. You'll see that the algorithm follows the same steps as when using the DFS button to do a search. When using Tree, however, the appropriate edge is darkened when the algorithm assigns it to the minimum spanning tree. When it's finished, the applet removes all the non-darkened lines, leaving only the minimum spanning tree. A final button press restores the original graph, in case you want to use it again.

## Java Code for the Minimum Spanning Tree

Here's the code for the `mst()` method:

```
while( !theStack.isEmpty() )       // until stack empty
   {                               // get stack top
   int currentVertex = theStack.peek();
   // get next unvisited neighbor
   int v = getAdjUnvisitedVertex(currentVertex);
   if(v == -1)                     // if no more neighbors
      theStack.pop();              //    pop it away
   else                            // got a neighbor
      {
      vertexList[v].wasVisited = true;  // mark it
      theStack.push(v);                 // push it
                                  // display edge
      displayVertex(currentVertex);     // from currentV
      displayVertex(v);                 // to v
      System.out.print(" ");
      }
   }  // end while(stack not empty)

   // stack is empty, so we're done
```

```
      for(int j=0; j<nVerts; j++)          // reset flags
         vertexList[j].wasVisited = false;

   }  // end mst()
```

As you can see, this code is very similar to `dfs()`. In the `else` statement, however, the current vertex and its next unvisited neighbor are displayed. These two vertices define the edge that the algorithm is currently traveling to get to a new vertex, and it's these edges that make up the minimum spanning tree.

In the `main()` part of the `mst.java` program, we create a graph by using these statements:

```
Graph theGraph = new Graph();
theGraph.addVertex('A');    // 0  (start for mst)
theGraph.addVertex('B');    // 1
theGraph.addVertex('C');    // 2
theGraph.addVertex('D');    // 3
theGraph.addVertex('E');    // 4

theGraph.addEdge(0, 1);     // AB
theGraph.addEdge(0, 2);     // AC
theGraph.addEdge(0, 3);     // AD
theGraph.addEdge(0, 4);     // AE
theGraph.addEdge(1, 2);     // BC
theGraph.addEdge(1, 3);     // BD
theGraph.addEdge(1, 4);     // BE
theGraph.addEdge(2, 3);     // CD
theGraph.addEdge(2, 4);     // CE
theGraph.addEdge(3, 4);     // DE
```

The graph that results is the one shown in Figure 13.10-a. When the `mst()` method has done its work, only four edges are left, as shown in Figure 13.10-b. Here's the output from the `mst.java` program:

```
Minimum spanning tree: AB BC CD DE
```

As we noted, this is only one of many possible minimum scanning trees that can be created from this graph. Using a different starting vertex, for example, would result in a different tree. So would small variations in the code, such as starting at the end of the `vertexList[]` instead of the beginning in the `getAdjUnvisitedVertex()` method.

## The `mst.java` Program

Listing 13.3 shows the `mst.java` program. It's similar to `dfs.java`, except for the `mst()` method and the graph created in `main()`.

### Listing 13.3 The mst.java Program

```
// mst.java

// demonstrates minimum spanning tree

// to run this program: C>java MSTApp
```

```
import java.awt.*;

////////////////////////////////////////////////////////////////

class StackX
   {
   private final int SIZE = 20;
   private int[] st;
   private int top;
   public StackX()            // constructor
      {
      st = new int[SIZE];     // make array
      top = -1;
      }
   public void push(int j)    // put item on stack
      { st[++top] = j; }
   public int pop()           // take item off stack
      { return st[top--]; }
   public int peek()          // peek at top of stack
      { return st[top]; }
   public boolean isEmpty()   // true if nothing on stack
      { return (top == -1); }
   }

////////////////////////////////////////////////////////////////

class Vertex
   {
   public char label;         // label (e.g. 'A')
   public boolean wasVisited;

// ---------------------------------------------------------
-
   public Vertex(char lab)    // constructor
      {
      label = lab;
      wasVisited = false;
      }

// ---------------------------------------------------------
-
   }  // end class Vertex

////////////////////////////////////////////////////////////////

class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // list of vertices
   private int adjMat[][];      // adjacency matrix
          // current number of vertices
   private StackX theStack;
```

```
// ------------------------------------------------------------
-
   public Graph()               // constructor
      {
      vertexList = new Vertex[MAX_VERTS];
                                          // adjacency matrix
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // set adjacency
         for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
            adjMat[j][k] = 0;
      theStack = new StackX();
      }  // end constructor

// ------------------------------------------------------------
-
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }

// ------------------------------------------------------------
-
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      adjMat[end][start] = 1;
      }

// ------------------------------------------------------------
-
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }

// ------------------------------------------------------------
-
   public void mst()  // minimum spanning tree (depth first)
      {                                // start at 0
      vertexList[0].wasVisited = true;   // mark it
      theStack.push(0);                  // push it

      while( !theStack.isEmpty() )       // until stack empty
         {                               // get stack top
         int currentVertex = theStack.peek();
         // get next unvisited neighbor
         int v = getAdjUnvisitedVertex(currentVertex);
         if(v == -1)                     // if no more
neighbors
            theStack.pop();              //    pop it away
         else                            // got a neighbor
            {
```

```
            vertexList[v].wasVisited = true;  // mark it
            theStack.push(v);                 // push it
                                          // display edge
            displayVertex(currentVertex);     // from currentV
            displayVertex(v);                 // to v
            System.out.print(" ");
            }
         }  // end while(stack not empty)

         // stack is empty, so we're done
         for(int j=0; j<nVerts; j++)          // reset flags
            vertexList[j].wasVisited = false;
      }  // end tree

// ------------------------------------------------------------
-
   // returns an unvisited vertex adj to v
   public int getAdjUnvisitedVertex(int v)
      {
      for(int j=0; j<nVerts; j++)
         if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
            return j;
      return -1;
      }  // end getAdjUnvisitedVert()

// ------------------------------------------------------------
-
   }  // end class Graph

////////////////////////////////////////////////////////////////

class MSTApp
   {
   public static void main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A');    // 0  (start for mst)
      theGraph.addVertex('B');    // 1
      theGraph.addVertex('C');    // 2
      theGraph.addVertex('D');    // 3
      theGraph.addVertex('E');    // 4

      theGraph.addEdge(0, 1);     // AB
      theGraph.addEdge(0, 2);     // AC
      theGraph.addEdge(0, 3);     // AD
      theGraph.addEdge(0, 4);     // AE
      theGraph.addEdge(1, 2);     // BC
      theGraph.addEdge(1, 3);     // BD
      theGraph.addEdge(1, 4);     // BE
      theGraph.addEdge(2, 3);     // CD
      theGraph.addEdge(2, 4);     // CE
      theGraph.addEdge(3, 4);     // DE
```

```
        System.out.print("Minimum spanning tree: ");
        theGraph.mst();             // minimum spanning tree
        System.out.println();
        }  // end main()
    }  // end class MSTApp

////////////////////////////////////////////////////////////////
```

## Topological Sorting with Directed Graphs

Topological sorting is another operation that can be modeled with graphs. It's useful in situations in which items or events must be arranged in a specific order. Let's look at an example.

## An Example: Course Prerequisites

In high school and college, students find (sometimes to their dismay) that they can't take just any course they want. Some courses have prerequisites—other courses that must be taken first. Indeed, taking certain courses may be a prerequisite to obtaining a degree in a certain field. Figure 13.11 shows a somewhat fanciful arrangement of courses necessary for graduating with a degree in mathematics.

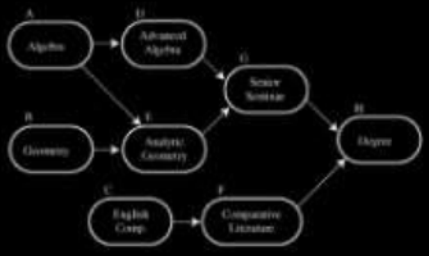

**Figure 13.11:** Course prerequisites

To obtain your degree, you must complete the Senior Seminar and (because of pressure from the English Department) Comparative Literature. But you can't take Senior Seminar without having already taken Advanced Algebra and Analytic Geometry, and you can't take Comparative Literature without taking English Composition. Also, you need Geometry for Analytic Geometry, and Algebra for both Advanced Algebra and Analytic Geometry.

## Directed Graphs

As the figure shows, a graph can represent this sort of arrangement. However, the graph needs a feature we haven't seen before: The edges need to have a *direction*.

**Figure 13.12:** A small directed graph

When this is the case, the graph is called a *directed* graph. In a directed graph you can only proceed one way along an edge. The arrows in Figure 13.11 show the direction of

the edges.

In a program, the difference between a non-directed graph and a directed graph is that an edge in a directed graph has only one entry in the adjacency matrix. Figure 13.12 shows a small directed graph; Table 13.5 shows its adjacency matrix.

**Table 13.5: Adjacency Matrix for Small Directed Graph**

| | A | B | C |
|---|---|---|---|
| A | 0 | 1 | 0 |
| B | 0 | 0 | 1 |
| C | 0 | 0 | 0 |

Each edge is represented by a single 1. The row labels show where the edge starts, and the column labels show where it ends. Thus, the edge from A to B is represented by a single 1 at row A column B. If the directed edge were reversed so that it went from B to A, there would be a 1 at row B column A instead.

For a non-directed graph, as we noted earlier, half of the adjacency matrix mirrors the other half, so half the cells are redundant. However, for a weighted graph, every cell in the adjacency matrix conveys unique information.

For a directed graph, the method that adds an edge thus needs only a single statement,

```
public void addEdge(int start, int end)  // directed graph
   {
   adjMat[start][end] = 1;

   }
```

instead of the two statements required in a non-directed graph.

If you use the adjacency-list approach to represent your graph, then A has B in its list but—unlike a non-directed graph—B does not have A in its list.

## Topological Sorting

Imagine that you make a list of all the courses necessary for your degree, using Figure 13.11 as your input data. You then arrange the courses in the order you need to take them. Obtaining your degree is the last item on the list, which might look like this:

BAEDGCFH

Arranged this way, the graph is said to be *topologically sorted*. Any course you must take before some other course occurs before it in the list.

Actually, many possible orderings would satisfy the course prerequisites. You could take the English courses C and F first, for example:

CFBAEDGH

This also satisfies all the prerequisites. There are many other possible orderings as well. When you use an algorithm to generate a topological sort, the approach you take and the details of the code determine which of various valid sortings are generated.

Topological sorting can model other situations besides course prerequisites. Job scheduling is an important example. If you're building a car, you want to arrange things so that brakes are installed before the wheels and the engine is assembled before it's bolted onto the chassis. Car manufacturers use graphs to model the thousands of operations in the manufacturing process, to ensure that everything is done in the proper order.

Modeling job schedules with graphs is called *critical path analysis*. Although we don't show it here, a weighted graph (discussed in the next chapter) can be used, which allows the graph to include the time necessary to complete different tasks in a project. The graph can then tell you such things as the minimum time necessary to complete the project.

## The GraphD Workshop Applet

The GraphD Workshop applet models directed graphs. This applet operates in much the same way as GraphN but provides a dot near one end of each edge to show which direction the edge is pointing. Be careful: The direction you drag the mouse to create the edge determines the direction of the edge. Figure 13.13 shows the GraphD workshop applet used to model the course-prerequisite situation of Figure 13.11.

The idea behind the topological sorting algorithm is unusual but simple. Two steps are necessary:

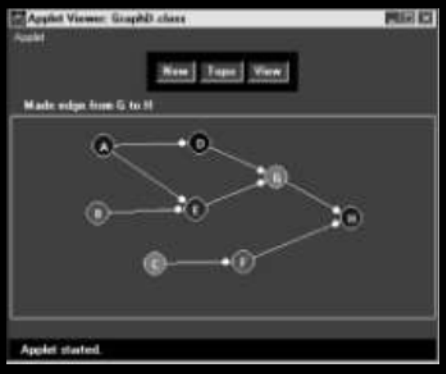


**Figure 13.13:** The GraphD Workshop applet

**REMEMBER**

Step 1: Find a vertex that has no successors.

The successors to a vertex are those vertices that are directly "downstream" from it—that is, connected to it by an edge that points in their direction. If there is an edge pointing from A to B, then B is a successor to A. In Figure 13.11, the only vertex with no successors is H.

**REMEMBER**

Step 2: Delete this vertex from the graph, and insert its label at the beginning of a list.

Steps 1 and 2 are repeated until all the vertices are gone. At this point, the list shows the vertices arranged in topological order.

You can see the process at work by using the GraphD applet. Construct the graph of Figure 13.11 (or any other graph, if you prefer) by double-clicking to make vertices and dragging to make edges. Then repeatedly click the Topo button. As each vertex is removed, its label is placed at the beginning of the list below the graph.

Deleting a vertex may seem like a drastic step, but it's the heart of the algorithm. The algorithm can't figure out the second vertex to remove until the first vertex is gone. If you need to, you can save the graph's data (the vertex list and the adjacency matrix) elsewhere and restore it when the sort is completed, as we do in the GraphD applet.

The algorithm works because if a vertex has no successors, it must be the last one in the topological ordering. As soon as it's removed, one of the remaining vertices must have no successors, so it will be the next-to-last one in the ordering, and so on.

The topological sorting algorithm works on unconnected graphs as well as connected graphs. This models the situation in which you have two unrelated goals, such as getting a degree in mathematics and at the same time obtaining a certificate in first aid.

## Cycles and Trees

One kind of graph the topological-sort algorithm cannot handle is a graph with *cycles*. What's a cycle? It's a path that ends up where it started. In Figure 13.14 the path B-C-D-B forms a cycle. (Notice that A-B-C-A is not a cycle because you can't go from C to A.)

**Figure 13.14:** Graph with a cycle

A cycle models the Catch-22 situation (which some students claim to have actually encountered at certain institutions), where course B is a prerequisite for course C, C is a prerequisite for D, and D is a prerequisite for B.

A graph with no cycles is called a *tree*. The binary and multiway trees we saw earlier in this book are trees in this sense. However, the trees that arise in graphs are more general than binary and multiway trees, which have a fixed maximum number of child nodes. In a graph, a vertex in a tree can be connected to any number of other vertices, provided that no cycles are created.

A topological sort is carried out on a directed graph with no cycles. Such a graph is called a *directed, acyclic graph*, often abbreviated DAG.

## Java Code

Here's the Java code for the `topo()` method, which carries out the topological sort:

```
   public void topo()            // toplogical sort
      {
```

```
      int orig_nVerts = nVerts;  // remember how many verts

      while(nVerts > 0)          // while vertices remain,
         {
         // get a vertex with no successors, or -1
         int currentVertex = noSuccessors();
         if(currentVertex == -1)       // must be a cycle
            {
            System.out.println("ERROR: Graph has cycles");
            return;
            }
         // insert vertex label in sorted array (start at end)
         sortedArray[nVerts-1] = vertexList[currentVertex].label;

         deleteVertex(currentVertex);  // delete vertex
         }  // end while

      // vertices all gone; display sortedArray
      System.out.print("Topologically sorted order: ");
      for(int j=0; j<orig_nVerts; j++)
         System.out.print( sortedArray[j] );
      System.out.println("");

      }  // end topo
```

The work is done in the `while` loop, which continues until the number of vertices is reduced to 0. Here are the steps involved:

1. Call `noSuccessors()` to find any vertex with no successors.

2. If such a vertex is found, put the vertex label at the end of `sortedArray[]` and delete the vertex from the graph.

If an appropriate vertex isn't found, the graph must have a cycle.

The last vertex to be removed appears first on the list, so the vertex label is placed in `sortedArray` starting at the end and working toward the beginning, as `nVerts` (the number of vertices in the graph) gets smaller.

If vertices remain in the graph but all of them have successors, the graph must have a cycle, and the algorithm displays a message and quits. Normally, however, the `while` loop exits, and the list from `sortedArray` is displayed, with the vertices in topologically sorted order.

The `noSuccessors()` method uses the adjacency matrix to find a vertex with no successors. In the outer `for` loop, it goes down the rows, looking at each vertex. For each vertex, it scans across the columns in the inner `for` loop, looking for a 1. If it finds one, it knows that that vertex has a successor, because there's an edge from that vertex to another one. When it finds a 1, it bails out of the inner loop so that the next vertex can be investigated.

Only if an entire row is found with no 1s do we know we have a vertex with no successors; in this case, its row number is returned. If no such vertex is found, –1 is returned. Here's the `noSuccessors()` method:

```
public int noSuccessors()  // returns vert with no successors
   {                       // (or -1 if no such verts)
   boolean isEdge;  // edge from row to column in adjMat

   for(int row=0; row<nVerts; row++)  // for each vertex,
      {
      isEdge = false;                 // check edges
      for(int col=0; col<nVerts; col++)
         {
         if( adjMat[row][col] > 0 )   // if edge to
            {                         // another,
            isEdge = true;
            break;                    // this vertex
            }                         //    has a successor
         }                            //    try another
      if( !isEdge )                   // if no edges,
         return row;                  //    has no successors
      }
   return -1;                         // no such vertex

   }  // end noSuccessors()
```

Deleting a vertex is straightforward except for a few details. The vertex is removed from the `vertexList[]` array, and the vertices above it are moved down to fill up the vacant position. Likewise, the row and column for the vertex are removed from the adjacency matrix, and the rows and columns above and to the right are moved down and to the left to fill the vacancies. This is carried out by the `deleteVertex()`, `moveRowUp()`, and `moveColLeft()` methods, which you can examine in the complete listing for `topo.java`. It's actually more efficient to use the adjacency-list representation of the graph for this algorithm, but that would take us too far afield.

The `main()` routine in this program calls on methods, similar to those we saw earlier, to create the same graph shown in Figure 13.10. The `addEdge()` method, as we noted, inserts a single number into the adjacency matrix because this is a directed graph. Here's the code for `main()`:

```
public static void main(String[] args)
   {
   Graph theGraph = new Graph();
   theGraph.addVertex('A');    // 0
   theGraph.addVertex('B');    // 1
   theGraph.addVertex('C');    // 2
   theGraph.addVertex('D');    // 3
   theGraph.addVertex('E');    // 4
   theGraph.addVertex('F');    // 5
   theGraph.addVertex('G');    // 6
   theGraph.addVertex('H');    // 7

   theGraph.addEdge(0, 3);     // AD
   theGraph.addEdge(0, 4);     // AE
   theGraph.addEdge(1, 4);     // BE
   theGraph.addEdge(2, 5);     // CF
   theGraph.addEdge(3, 6);     // DG
   theGraph.addEdge(4, 6);     // EG
```

```
      theGraph.addEdge(5, 7);      // FH
      theGraph.addEdge(6, 7);      // GH

      theGraph.topo();             // do the sort

      }  // end main()
```

Once the graph is created, `main()` calls `topo()` to sort the graph and display the result. Here's the output:

```
Topologically sorted order: BAEDGCFH
```

Of course, you can rewrite `main()` to generate other graphs.

## The Complete topo.java Program

You've seen most of the routines in `topo.java` already. Listing 13.4 shows the complete program.

### Listing 13.4 The topo.java Program

```
// topo.java
// demonstrates topological sorting
// to run this program: C>java TopoApp
import java.awt.*;
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // label (e.g. 'A')

   public Vertex(char lab)   // constructor
      { label = lab; }
   }  // end class Vertex

////////////////////////////////////////////////////////////////

class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // list of vertices
   private int adjMat[][];      // adjacency matrix
   private int nVerts;          // current number of vertices
   private char sortedArray[];

// ------------------------------------------------------------
-
   public Graph()               // constructor
      {
      vertexList = new Vertex[MAX_VERTS];
                                          // adjacency matrix
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // set adjacency
```

```
         for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
            adjMat[j][k] = 0;
      sortedArray = new char[MAX_VERTS];  // sorted vert labels
      }  // end constructor

// ------------------------------------------------------------
-
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }

// ------------------------------------------------------------
-
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      }

// ------------------------------------------------------------
-
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }

// ------------------------------------------------------------
-
   public void topo()  // toplogical sort
      {
      int orig_nVerts = nVerts;  // remember how many verts

      while(nVerts > 0)  // while vertices remain,
         {
         // get a vertex with no successors, or -1
         int currentVertex = noSuccessors();
         if(currentVertex == -1)       // must be a cycle
            {
            System.out.println("ERROR: Graph has cycles");
            return;
            }
         // insert vertex label in sorted array (start at end)
         sortedArray[nVerts-1] =
vertexList[currentVertex].label;

         deleteVertex(currentVertex);  // delete vertex
         }  // end while

      // vertices all gone; display sortedArray
      System.out.print("Topologically sorted order: ");
      for(int j=0; j<orig_nVerts; j++)
         System.out.print( sortedArray[j] );
      System.out.println("");
      }  // end topo
```

```
------------------
   public int noSuccessors()  // returns vert with no
successors
      {                       // (or -1 if no such verts)
      boolean isEdge;  // edge from row to column in adjMat

      for(int row=0; row<nVerts; row++)  // for each vertex,
         {
         isEdge = false;                 // check edges
         for(int col=0; col<nVerts; col++)
            {
            if( adjMat[row][col] > 0 )   // if edge to
               {                         // another,
               isEdge = true;
               break;                    // this vertex
               }                         //    has a successor
            }                            //    try another
         if( !isEdge )                   // if no edges,
            return row;                  //    has no
successors
         }
      return -1;                         // no such vertex
      }  // end noSuccessors()

// ------------------
   public void deleteVertex(int delVert)
      {
      if(delVert != nVerts-1)      // if not last vertex,
         {                         // delete from vertexList
         for(int j=delVert; j<nVerts-1; j++)
            vertexList[j] = vertexList[j+1];
                                   // delete row from adjMat
         for(int row=delVert; row<nVerts-1; row++)
            moveRowUp(row, nVerts);
                                   // delete col from adjMat
         for(int col=delVert; col<nVerts-1; col++)
            moveColLeft(col, nVerts-1);
         }
      nVerts--;                    // one less vertex
      }  // end deleteVertex

// ------------------
   private void moveRowUp(int row, int length)
      {
      for(int col=0; col<length; col++)
         adjMat[row][col] = adjMat[row+1][col];
      }

// ------------------
   private void moveColLeft(int col, int length)
      {
      for(int row=0; row<length; row++)
```

```
            adjMat[row][col] = adjMat[row][col+1];
         }

   // ------------------------------------------------------------
   -
      }  // end class Graph

   ////////////////////////////////////////////////////////////////

   class TopoApp
      {
      public static void main(String[] args)
         {
         Graph theGraph = new Graph();
         theGraph.addVertex('A');    // 0
         theGraph.addVertex('B');    // 1
         theGraph.addVertex('C');    // 2
         theGraph.addVertex('D');    // 3
         theGraph.addVertex('E');    // 4
         theGraph.addVertex('F');    // 5
         theGraph.addVertex('G');    // 6
         theGraph.addVertex('H');    // 7

         theGraph.addEdge(0, 3);     // AD
         theGraph.addEdge(0, 4);     // AE
         theGraph.addEdge(1, 4);     // BE
         theGraph.addEdge(2, 5);     // CF
         theGraph.addEdge(3, 6);     // DG
         theGraph.addEdge(4, 6);     // EG
         theGraph.addEdge(5, 7);     // FH
         theGraph.addEdge(6, 7);     // GH

         theGraph.topo();            // do the sort
         }  // end main()
      }  // end class TopoApp

   ////////////////////////////////////////////////////////////////
```

In the next chapter, we'll see what happens when edges are given a weight as well as a direction.

## Summary

- Graphs consist of vertices connected by edges.
- Graphs can represent many real-world entities, including airline routes, electrical circuits, and job scheduling.
- Search algorithms allow you to visit each vertex in a graph in a systematic way. Searches are the basis of several other activities.
- The two main search algorithms are depth-first search (DFS) and breadth-first search (BFS).

- The depth-first search algorithm can be based on a stack; the breadth-first search algorithm can be based on a queue.
- A minimum spanning tree (MST) consists of the minimum number of edges necessary to connect all a graph's vertices.
- A slight modification of the depth-first search algorithm on an unweighted graph yields its minimum spanning tree.
- In a directed graph, edges have a direction (often indicated by an arrow).
- A topological sorting algorithm creates a list of vertices arranged so that a vertex A precedes a vertex B in the list if there's a path from A to B.
- A topological sort can be carried out only on a DAG, a directed, acyclic (no cycles) graph.
- Topological sorting is typically used for scheduling complex projects that consist of tasks contingent on other tasks.

# Chapter 14: Weighted Graphs

## Overview

In the last chapter we saw that a graph's edges can have direction. In this chapter we'll explore another edge feature: weight. For example, if vertices in a weighted graph represent cities, the weight of the edges might represent distances between the cities, or costs to fly between them, or the number of automobile trips made annually between them (a figure of interest to highway engineers).

When we include weight as a feature of a graph's edges, some interesting and complex questions arise. What is the minimum spanning tree for a weighted graph? What is the shortest (or cheapest) distance from one vertex to another? Such questions have important applications in the real world.

We'll first examine a weighted but non-directed graph and its minimum spanning tree. In the second half of this chapter we'll examine graphs that are both directed and weighted, in connection with the famous Dijkstra's Algorithm, used to find the shortest path from one vertex to another.

## Minimum Spanning Tree with Weighted Graphs

To introduce weighted graphs we'll return to the question of the minimum spanning tree. Creating such a tree is a bit more complicated with a weighted graph than with an unweighted one. When all edges are the same length it's fairly straightforward—as we saw in the last chapter—for the algorithm to choose one to add to the minimum spanning tree. But when edges can have different weights, some arithmetic is needed to choose the right one.

## An Example: Cable TV in the Jungle

Suppose we want to install a cable television line that connects six towns in the mythical country of Magnaguena. Five links will connect the six cities, but which five links should they be? The cost of connecting each pair of cities varies, so we must pick the route carefully to minimize the overall cost.

Figure 14.1 shows a weighted graph with 6 vertices, representing the towns Ajo, Bordo, Colina, Danza, Erizo, and Flor. Each edge has a weight, shown by a number alongside